□ Topic of This Presentation: HAL

■ HAL is a new constraint logic programming language.

■ What does HAL mean ?

- It was the name of the computer in *Stanley Kubrick*'s cult science fiction film, 2001: A Space Odyssey

■ Developer Team Hopes

"We're hoping that CLP and HAL will eventually be to the industry and problem-solving what Java has already become to network programming. Judging by industry, it could well be."

□ Extending the Constraint Solver

■ Consider extending *CLP(R)* to allow complex numbers.

- A complex number $x + iy$ is represented by the term `c(x,y)`.
- Equations over complex numbers are provided by term equality "=".

```
1   c_add(c(R1, I1), c(R2, I2), c(R3, I3)) :-
2        R3 = R1+R2, I3 = I1+I2.
3
4   c_mult(c(R1, I1), c(R2, I2), c(R3, I3)) :-
5        R3 = R1*R2 - I1+I2, I3 = R1*I2 + R2*I1.
6
7   c_ne(X, Y):- not(X=Y).
```

■ Limitations

- No overloading of function and primitive constraints names used by the solver.
- Not possible to directly define functions.
- For certain modes of usage, the extended solver may not be well-behaved or even not terminate.
- The solver is a black box.

□ Extending the Constraint Solver

■ Glass Box Solvers

Ex: *clp(FD)* proposed by Codognet and Diaz

- Based on the primitive constraint `X in r`
  - `X in 3..20`
  - `X in min(Y)..40`
  - `X in dom(Y):1..max(Z)`
- The semantics of `X in r`: adding `X in r` to the store S consists in updating `X` and in reactivating all constraints depending on `X` (propagation).
- User define constraints are supported.

```
1   'x=y+c'(X,Y,C):- X in min(Y)+C..max(Y)+C,
2                    Y in min(X)-C..max(X)-C.
3
4   %%% A non-linear equation
5   'xx=z'(X,Z):- X in sqrt_e(min(Z))..sqrt_d(max(Z)),
6                 Z in min(X)*min(X)..max(X)*max(X).
```

□ Major Drawbacks of Current *CLP* Languages

- Efficiency.
- Robustness.
- Flexible choice of constraint solvers ("plug and play" with different solvers, extend an existing solver, create a hybrid solver by combining solvers, write a new solver).

■ HAL tries to achieve these design objectives.

□ Main Features of HAL

- Type, mode and determinism declarations.
  - Efficiency.
  - Robustness.
- Solvers are modules with well-defined interfaces.
  - "Plug and play" experimentation with different solvers.
- Dynamic scheduling of goals.
  - Writing new constraint solvers, extending a solver and combining different solvers.

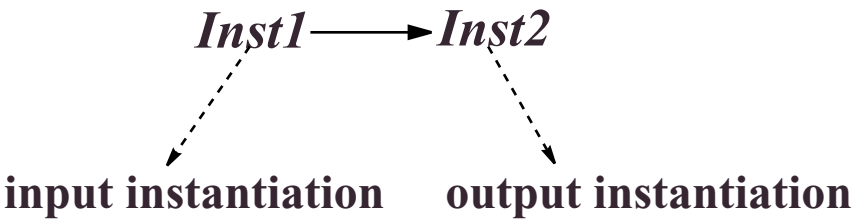
□ The HAL Language

- Standard *CLP* syntax.
- Ex: Implementation of a polymorphic stack using lists.

```
 1   :- typedef list(T) -> ([] ; [T|list(T)]).
 2
 3   :- instdef elist -> [].
 4   :- instdef list(I) -> ([] ; [I|list(I)]).
 5   :- instdef nelist -> [ground|list(ground)].
 6
 7   :- modedef out(i) -> (new -> I).
 8   :- modedef in(I) -> (I -> I).
 9
10
11   :- pred push(list(T), T, list(T)).          % type decl
12   :- mode push(in, in, out(nelist)) is det.   % mode decl
13   push(S0, E, S1):- S1 = [E|S0].
14
15   :- pred pop(list(T), T, list(T)).
16   :- mode pop(in, out, out) is semidet.
17   :- mode pop(in(nelist), out, out) is det.
18   pop(S0, E, S1):- S0 = [E|S1].
19
20   :- pred empty(list(T)).
21   :- mode empty(in) is semidet.
22   :- mode empty(out(elist)) is det.
23   empty(S):- S = [].
```

□ The HAL Language

- Polymorphic type definitions are allowed (ex: `list(T)`).
- Instantiation definitions.
  - Base instantiations for a variable
    * `new`
    * `old`
    * `ground`
  - Parametric instantiation definitions are allowed (ex: `list(I)`).
- Mode definitions.



- Standard modes: `oo`, `no`, `og`, `gg=in`, `ng=out`.
- Parametric mode definitions are allowed (ex: `out(I)`, `in(I)`).

- Predicates/functions are preceded by type, mode and determinism declarations.

□ Modes Improve Efficiency

■ Ex. consider the following program written in a standard *CLP* language.

```
1   and(0,0,0).
2   and(0,1,0).
3   and(1,0,0).
4   and(1,1,1).
5
6   bnot(0,1).
7   bnot(1,0).
8
9   p :- and(X,Y,Z), and(Y, Z, T), bnot(Y, 0), bnot(T, 0).
```

- Since literals are selected from left to right, predicate `p` uses "deep" backtracking.
- The derivation tree for goal `p` is non-deterministic and has 31 states.

□ Modes Improve Efficiency

■ Ex. Consider the same program written in HAL.

```
 1   :- typedef bool -> (0 ; 1).
 2
 3   :- pred and(bool, bool, bool).
 4   :- mode and(in, in, out).
 5   :- mode and(in, out, in).
 6   :- mode and(out, in, in).
 7   and(0,0,0).
 8   and(0,1,0).
 9   and(1,0,0).
10   and(1,1,1).
11
12   :- pred bnot(bool, bool).
13   :- mode and(in, out).
14   :- mode and(out, in).
15   bnot(0,1).
16   bnot(1,0).
17
18   p :- and(X,Y,Z), and(Y, Z, T), bnot(Y, 0), bnot(T, 0).
```

- Using the information contained in the mode declarations, the HAL compiler re-orders the literals for predicate `p`.

```
1   p:- bnot(Y, 0), bnot(T, 0), and(Y, Z, T), and(X,Y,Z).
```

- The derivation tree for `p` is deterministic and has 13 states.

□ Modes Improve Efficiency

■ Ex: Consider the following code added to the stack example.

```
1   :- pred dupl(list(T), list(T)).
2   :- mode dupl(in(nelist), out(nelist)) is det.
3   dupl(S0, S):- S = [], S = [].
4   dupl(S0, S):- pop(S0, A, S1), push(S0, A, S).
```

The compiler generates the following code.

```
1   dupl-mode1(S0, S):- fail.
2   dupl-mode1(S0, S):- pop_mode2(S0, A, S1),
3                       push_mode1(S0, A, S).
```

- The compiler creates a specialized *procedure* for each mode:

  multi-variant specialization

- Both modes for `pop/3` are schedulable. The compiler chooses the mode with more specific calling instantiations.

- The first rule is removed ⇒ a choice point is eliminated during compilation.

□ Constraint Solvers as Modules

■ Ex: A module for handling complex numbers

```
 1   :- module complex.
 2   :- import simplex.
 3
 4   :- export_abstract typedef complex -> c(cfloat, cfloat).
 5
 6   :- export pred cx(cfloat, cfloat, complex).
 7   :-        mode cx(in, in, out) is det.
 8   :-        mode cx(out, out, in) is det.
 9   :-        mode cx(oo, oo, oo) is semidet.
10   cx(X, Y, c(X,Y)).
11
12   :- export func complex + complex --> complex.
13   :-        mode in + in --> out is det.
14   :-        mode oo + oo --> oo is semidet.
15   c(X1, Y1) + c(X2, Y2) --> c(X1+X2, Y1+Y2).
16
17   :- export pred init(complex).
18   :-        mode init(no) is det.
19   init(c(X,Y)):- init(X), init(Y).
20
21   :- export_only pred complex = complex.
22   :-             mode oo = oo.
23   c(X1, Y1) = c(X2, Y2) :- X1 = X2, Y1 = Y2.
24
25   :- coerce coerce_cfloat_complex(cfloat) --> complex.
26   :- export func coerce_cfloat_complex(cfloat) --> complex.
27   :-        mode coerce_cfloat_complex(in) --> out is det.
28   coerce_cfloat_complex(X) --> c(X,X).
```

□ Constraint Solvers as Modules

- Overloading of standard arithmetic operators and relations is allowed.
- The equality predicate is required for all constraint solvers.
- Typically, a coercion function will be defined.
  - The compiler may automatically add calls to a coercion function.
- The `init` predicate is used to initialize a variable and its mode is `no`.
  - The compiler may automatically add calls to the initialization predicate in user code.

□ Herbrand Solvers

- Most HAL types correspond to terms in traditional *CLP* languages.
- In the base language, terms can only use restricted forms of the equality predicate. For example, the following equalities are not allowed.

  `f(X, a) = f(b, Z)`, `X` and `Z` `new` vars.

  `X = Z`, `X` and `Z` `old` vars.

- Conceptually, HAL provides a constraint solver for each term type defined by the programmer – Herbrand Solver.
  - Full equality is provided – Unification;
  - An `init/1` predicate is also provided.

```
1   :- typedef htype -> ... .
2   :- herbrand htype.
```

□ Dynamic Scheduling

- Literals are not necessarily processed in left-to-right order.
- HAL's delay construct

$$cond_1 \Rightarrow goal_1 \;||\; \cdots \;||\; cond_n \Rightarrow goal_n$$

$goal_i$ remains active and is reexecuted whenever the delay condition $cond_i$ becomes true.

■ Ex: Module `M` exports the delay conditions `lbc(V)`, `ubc(V)` and `fixed(V)`.

The user can extend M's solver to new constraints.

```
1   X <= Y :- (lbc(X) ==> upd_lb(Y, lb(X)) ||
2              fixed(X) ==> upd_lb(Y, val(X)), kill ||
3              ubc(Y) ==> upd_ub(X, ub(Y)) ||
4              fixed(Y) ==> upd_ub(X, val(Y)), kill).
```

(bounds propagation)

□ Exporting Types for Delay Constructs

- To support delayed goals, a module needs to define the types of delay conditions and export them.

```
 1   %%% declarations in M module
 2
 3   :- export_abstract typedef cint = ... .
 4
 5   :- export typedef dcon -> (ubc(cint) | lbc(cint) |
 6                              fixed(cint)).
 7   :- export delay dcon.
 8
 9   :- export_abstract typedef delay_id = ... .
10   :- export pred get_id(delay_id).
11   :-        mode get_id(out) is det.
12
13   :- export pred delay(list(dcon), delay_id, list(pred)).
14   :- mode delay(in, in,
15                 in(list(pred is semidet))) is semidet.
16
17   :- export pred kill(delay_id).
18   :-        mode kill(in) is det.
```

- By exporting types for delay constructs, it is possible to build a "glass box" solver.

□ Combining Constraint Solvers in HAL

```
 1   :- module combined.
 2
 3   :- import M.
 4   :- import cplex.
 5
 6   :- export_abstract typedef combint -> p(cint, ilpint).
 7
 8   :- export pred combint >= combint.
 9   :-        mode oo >= oo is semidet.
10   p(X1, Y1) >= p(X2, Y2) :- X1 >= X2, Y1 >= Y2.
11
12   :- export pred init(combint).
13   :-        mode init(no) is det.
14   %%% Y in [lb(X)..ub(X)]
15   init(p(X, Y)) :- init(X), init(Y),
16                    (lbc(X) ==> Y >= lb(X)||
17                     ubc(X) ==> Y <= ub(X) ||
18                     fixed(X) ==> Y = val(X), kill).
```

- Constraints in the hybrid solver cause the constraints to be sent to the underlying solvers.
- Communication between solvers is achieved by delayed goals created when a variable is initialized.

□ Combining Constraint Solvers in HAL

- The compiler translates the delay construct in module `combined` as

```
1   get_id(D),
2   delay([lbc(X), ubc(X), fixed(X)], D,
3         [Y >= lb(X), Y <= ub(X), (Y = val(X), kill(D))]).
```

- The module `combined` has the responsibility
  - to determine when a delay condition has become true;
  - to call the corresponding goal; and
  - to remove killed actions.

□ Still about HAL

- HAL programs may be compiled to either Mercury or SICStus Prolog.
- Mercury is an efficient logic programming language that supports type, mode and determinism declarations.
- Mercury does not support full unification.
- Mercury is compiled to C.
- With appropriate declarations, HAL and its Herbrand solver is almost as fast as Mercury.
- Without declarations, its efficiency is comparable to SICStus Prolog.
- Biggest performance improvement arises from mode declarations. Type and determinism declarations give moderate speed improvement.
- All declarations reduce space requirements.